ReactJS

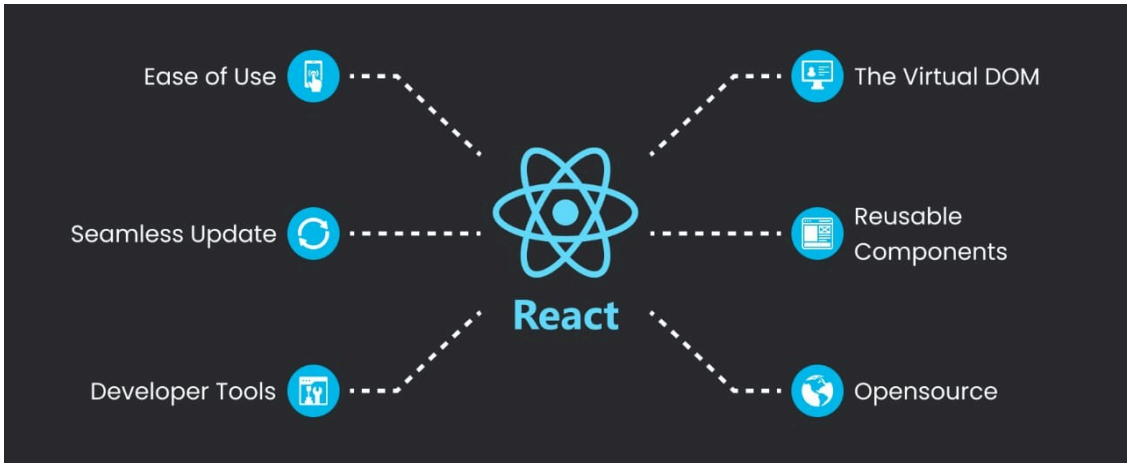
Projects :

Social Media

E-Commerce

Myntra Clone

Introduction to ReactJS



React is a JavaScript library developed at **Facebook** in 2011 for building dynamic and interactive **user interfaces**. It allows developers to create **single-page applications (SPAs)** where the page does not reload during navigation, ensuring a smoother user experience. React combines **JavaScript and XML-like syntax (JSX)** to create components and can be installed or used in two ways.

1. NPX :

Node Package Executor for setup for REACT

```
npx create-react-app my-app
```

2. VITE :

VITE is an Bundler , an easy and fast way for setup for REACT

```
 npm create vite@latest
```

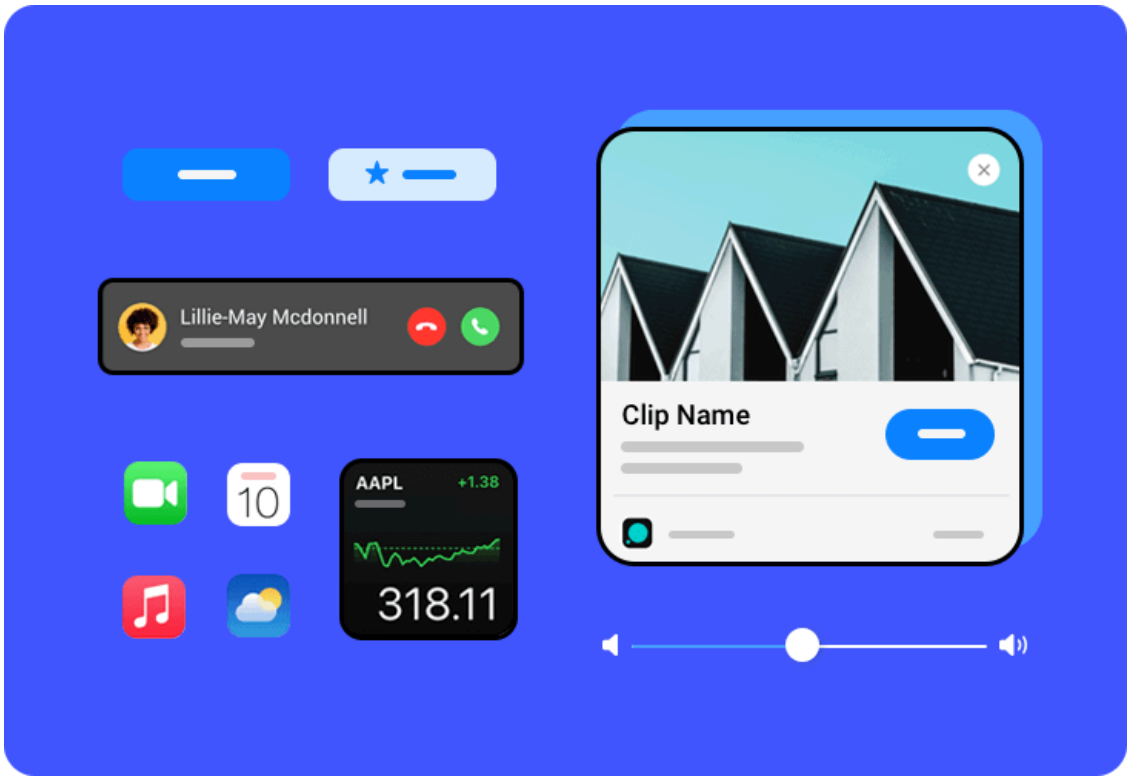
NOTE : We Need to install Node Modules separately while using VITE with command

```
npm install
```

Components

In React, components are the **building blocks** that help developers write **reusable**, **modular**, and **better-organized code**. A React application is structured as a **tree of components**,

with the App component serving as the root. This root component acts as the central hub, bringing all other components together to form the complete application.



Class Components

Class Components are **stateful**, meaning they can manage their own state and have access to **lifecycle methods** for controlling the component's behavior during its lifecycle. However, they tend to be more **verbose**, requiring additional boilerplate code, and are **not preferred** in modern React development.

Functional Components

Functional Components, initially designed to be stateless, have become more powerful with the introduction of **Hooks**, allowing them to manage state and side effects. They are **simpler, more concise**, and have gained **popularity**, becoming the standard for most React applications today.

Exporting Components

```
// Default export
export default ComponentName;

// Importing default export
import ComponentName from './ComponentPath';
```

Dynamic Components

Dynamic Content: With JSX, developers can easily define UIs that respond to changes in data or user interactions, making it ideal for creating dynamic content in applications.

JavaScript Expressions: Using curly braces `{}`, JSX allows embedding any **JavaScript expression** directly within the markup. This can

include variables, function calls, mathematical operations, or even conditional rendering. For example:

```
function CurrentTime() {
  let time = new Date();
  return (
    <div>
      <p>This is the clock that shows the time in bharat at all time.</p>
      <p className="fw-bold "> This is Current Time : {time.toLocaleTimeString()}
      and Date : {time.toLocaleDateString()}</p>
    </div>
  )

}

export default CurrentTime
```

Reusable Components

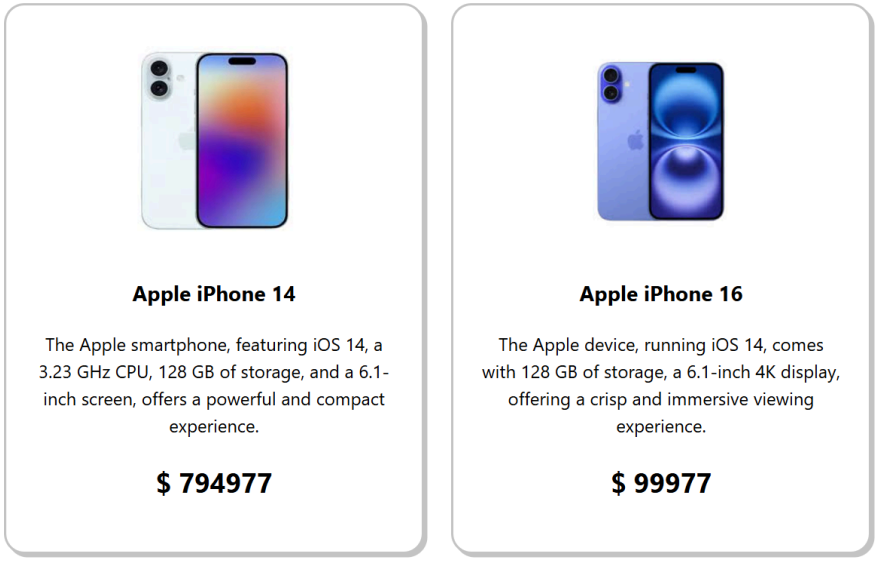
Modularity: Components are self-contained and modular, making it easy to reuse them across different parts of an application, reducing redundancy and improving organization.

Consistency: Reusing components ensures a consistent UI throughout the application, minimizing the risk of design discrepancies and maintaining a uniform user experience.

Efficiency: By avoiding code duplication, components significantly reduce development time and effort, enabling developers to focus on core functionalities instead of rewriting code.

Maintainability: Since components are reused, any updates or bug fixes made to a single component are automatically reflected across all instances where it is used, simplifying maintenance and updates.

```
import Card from './card'

function App() {

  return (
    <div>
      /* using card twice */
      <Card/>
      <Card/>
    </div>
  )
}

export default App
```



Including Bootstrap

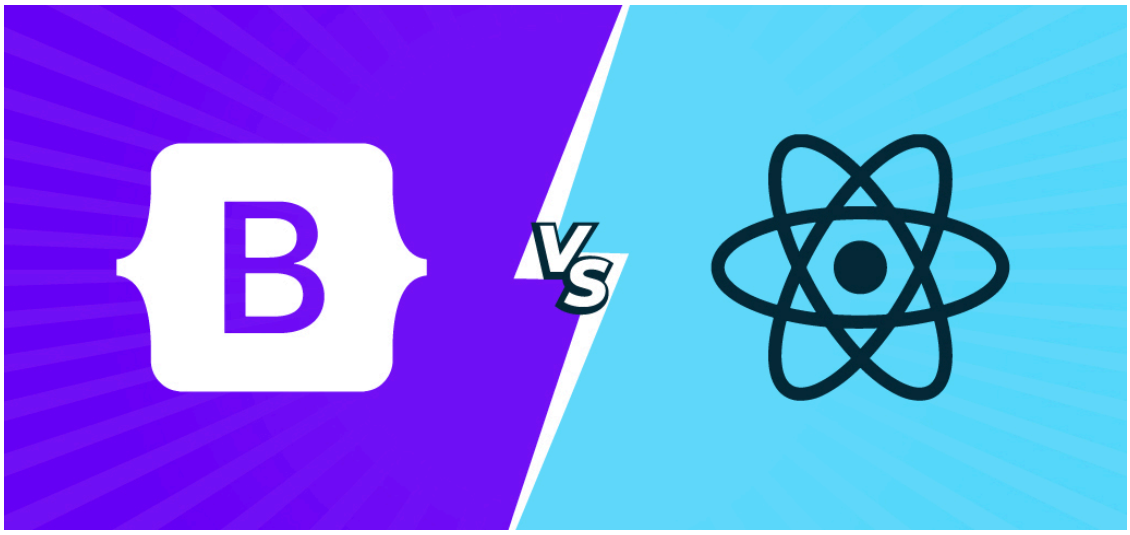
Responsive: Designed with a mobile-first approach, Bootstrap ensures that your application looks great on devices of all sizes, from phones to desktops.

Components: It includes a wide range of pre-styled elements, such as buttons, navbars, modals, and cards, making it easy to create polished interfaces quickly.

Customizable: Developers can modify default styles or use custom CSS to adapt the framework to match their specific design requirements.

Cross-Browser: Ensures a consistent look and feel across all major browsers, eliminating browser-specific styling issues.

Open-Source: As an open-source framework, it is free to use and benefits from robust community support for updates, plugins, and troubleshooting.



Install Bootstrap:

Use npm to install Bootstrap in your React project:

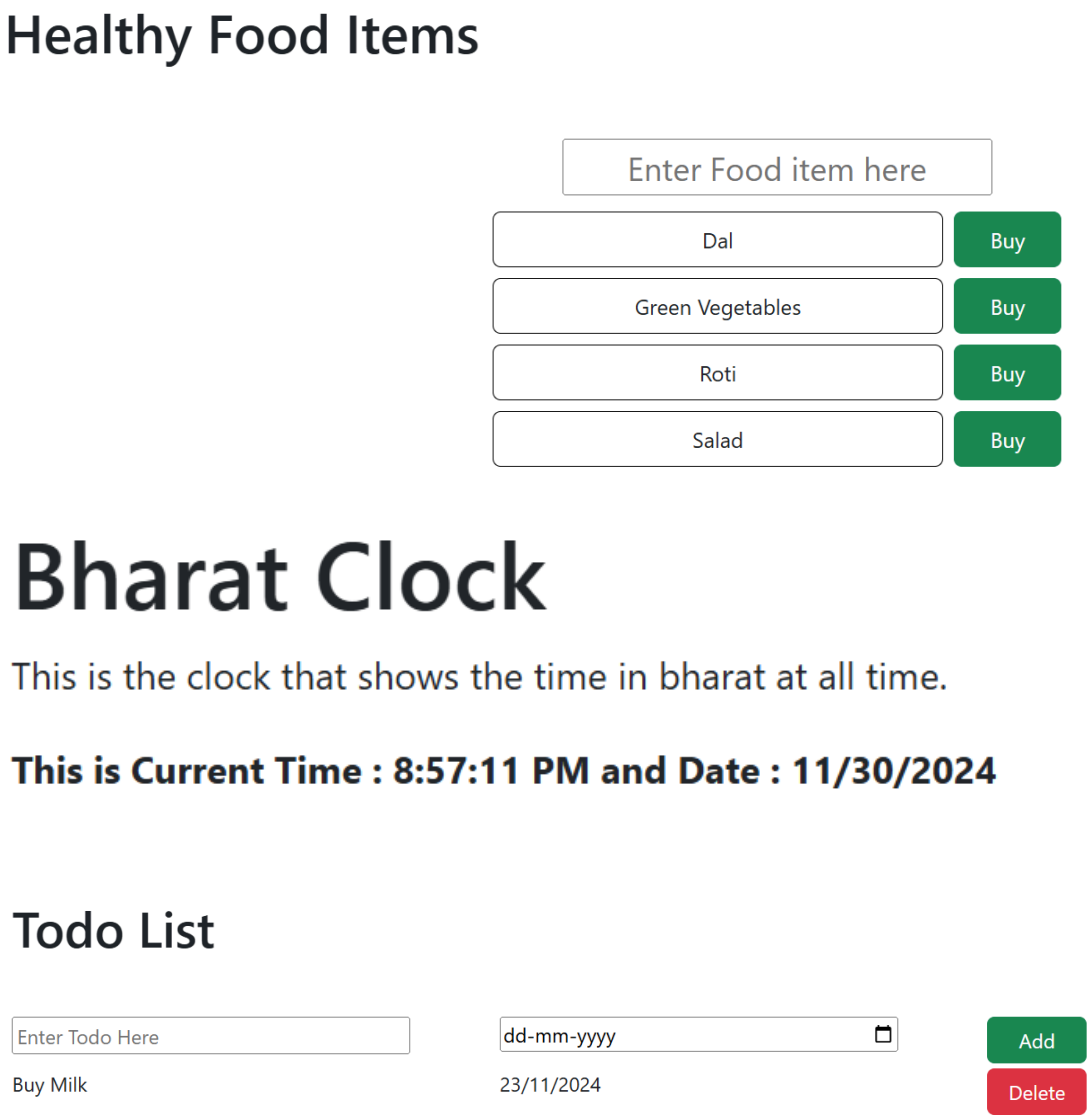
```
npm i bootstrap@5.3.2
```

Import Bootstrap (CSS):

```
import "bootstrap/dist/css/bootstrap.min.css";
```

Import Bootstrap (JS) :

```
import "bootstrap/dist/js/bootstrap.bundle.min.js";
```

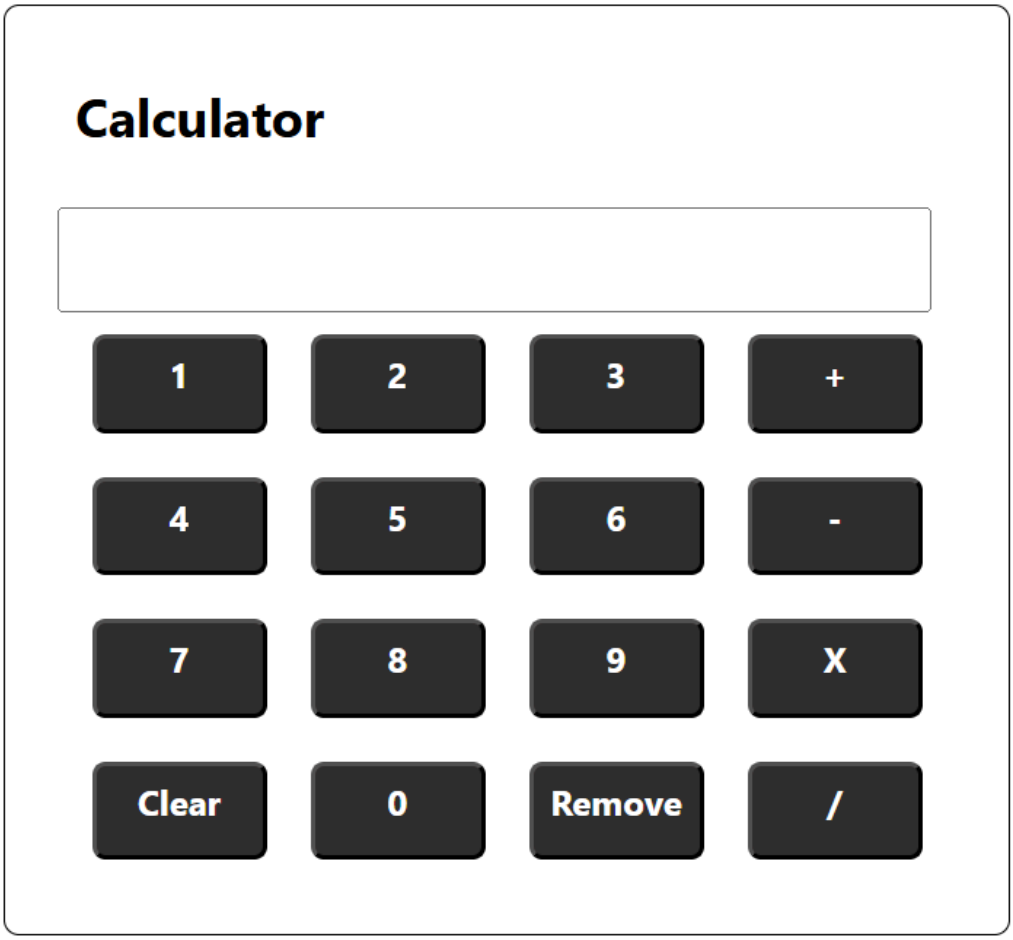


Map Method

You can render lists dynamically by iterating over an array and generating JSX for each item.

```
import CalBtn from './CalBtn'
function Calculator() {

  let buttons=["1","2","3","+","4","5","6","-","7","8","9","X",
          "Clear","0","Remove","/"]

  return (
    <div>
```

```
        <p className="fs-4 p-2 fw-bold">Calculator</p>
        <input type="text"/>
        <div>
        /* Mapping buttons List for generating keys / buttons */
          {buttons.map((btn) => ( <CalBtn key={btn}/> )}
        </div>
      </div>
    )
}

export default Calculator
```



Conditional Rendering

Conditional Rendering

- Displaying content based on certain conditions.
- Allows for dynamic user interfaces.

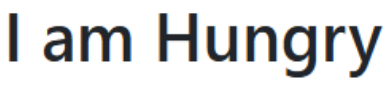
Methods

```
{foods.length===0 && <h3>I am Hungry</h3> } // Logical Opertation
```

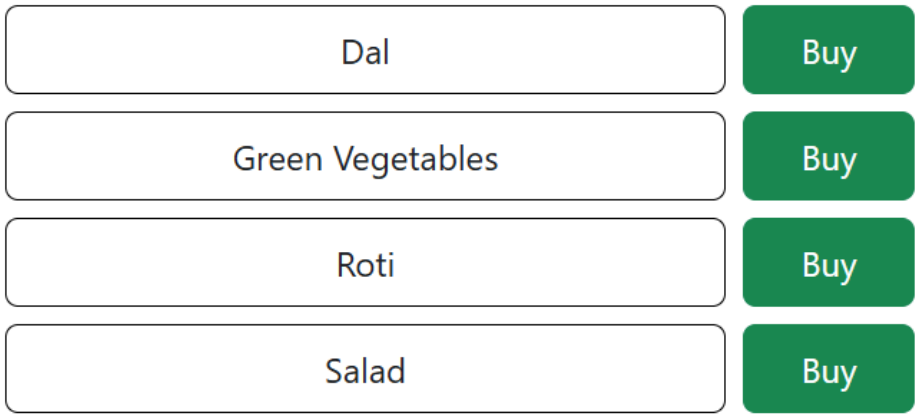
```
{foods.length === 0 ? <h3>I am Hungry</h3> : null} // Ternary Operation
```

```
function HealthFood() {

 foods = ['Dal', 'Green Vegetables', 'Roti', 'Salad']

 if (foods.length === 0) {
  return <h3>I am Hungry</h3>; // Display this if food is empty
 }

  return (
   <>
    <div>

     <ul>
      {foods.map((items) => (<Items key={items} food={items} />))}
     </ul>

    </div>
   </>
  )
}

export default HealthFood
```

if foods list is not empty



if foods list is not empty



Passing Data via Props

Short for properties: In React, props are short for "properties" and refer to the mechanism used for passing data between components.

Read-only by default: Props are immutable by default, meaning the child component receiving the prop cannot modify it.

Usage of Props

Pass data from parent to child component: Props allow you to send data from a parent component to a child component.

Defined as attributes in JSX: Props are passed as attributes when the component is used in JSX.

Key Points

Data flows one-way (downwards): Props always flow from parent to child components in React.

Props are immutable: The child component cannot modify the props passed to it.

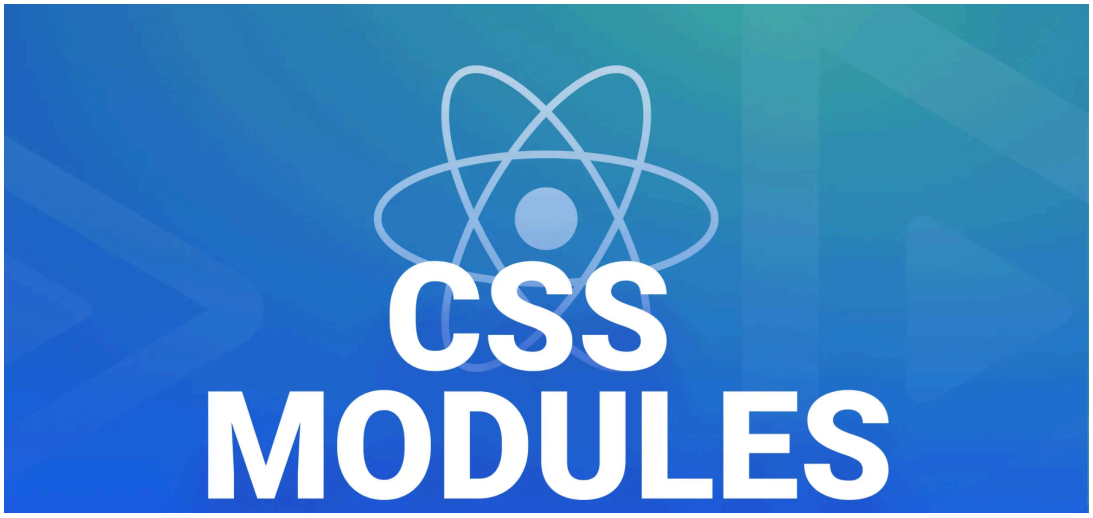
Used for communication between components: Props are the primary way that components communicate with each other.

```
// Parent Component
const App = () => {
  return <Header title="My App" />;
};

// Child Component
```

```
const Header = (props) => {
  return <h1>{props.title}</h1>;
};
```

CSS Modules

CSS Modules provide a way to write **localized class names** to avoid global conflicts. The styles are **scoped to individual components**, ensuring that each component has its own specific styles without affecting others. This helps in creating **component-specific styles** and **automatically generates unique class names** to prevent clashes. Using CSS Modules promotes **modular and maintainable CSS** by isolating styles to their respective components. You can also **use CSS Modules alongside global CSS** when necessary, allowing for flexibility in styling your application.



Calculator.module.css

```
.calculator {
    border: 1px solid black;
    padding: 24px;
    width: 40%;
    margin: 0 auto;
    border-radius: 7px;
    background-color: rgb(20, 20, 27);
```

```
    color:white;
}
```

Calculator.jsx

```
/*Importing CSS Module */
import Styles from './Calculator.module.css'

function Calculator() {

    return (
        <div className={Styles.calculator}>
            <p className="fs-4 p-2 fw-bold">Calculator</p>
            <input type="text"/>
           /* buttons */
        </div>
    )
}

export default Calculator
```

NOTE :

The **children** prop in React is used to pass elements into components, allowing for flexible and reusable component designs. It is commonly used in layout or container components and can be accessed using **props.children**. This prop can contain any type of content, such as strings, numbers, JSX, or other components. By using **children**, components become more composable and reusable, enhancing the overall flexibility of your app's design.

Container.jsx

```
function Container(props) {
    return (
        <div className={Styles.container}>
            {props.children}
        </div>
    )
}

export default Container
```

App.jsx

```
import Calculator from "./component/Calculator";
import Container from "./component/Container";
function App() {
 return (
  <>
   <Container>
    <Calculator />
   </Container>
  </>
 );
}

export default App;
```

Handling Events

React events use **camelCase [for eg : onClick]**

Uses **synthetic events**, not direct browser events.

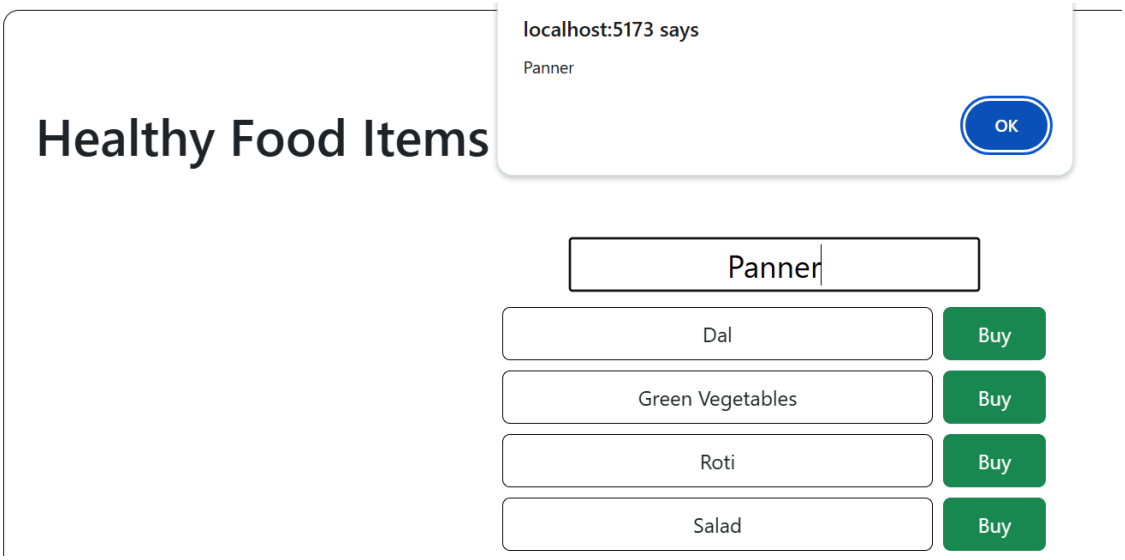
Event handlers can be **functions or arrow functions.**

Use **onChange** for controlled form inputs.

Avoid **inline arrow functions in JSX** for performance.

```
function HealthFood() {

/* Normal Function */
 function handleKeyDown(event){
  alert(event.target.value)
 }

/* Arrow Function */
 let handleKeyDown = (event) => {
  alert(event.target.value)
 }

 return (
  <>
   <h1 className="my-5">Healthy Food Items</h1>
   <div>
```

```
      <input type="text" onKeyDown={handleKeyDown}
       placeholder='Enter Food item here' />
     </div>

   </>
  )
}

export default HealthFood
```



Passing Functions via Props

Pass dynamic **behaviour** between components.

Enables upward communication from **child to parent.**

Commonly used for **event handling.**

Parent defines a function, child invokes it.

```
// Parent Component
const Parent = () => {
  const handleClick = () => {
    alert("Button clicked in the child component!");
  };
```

```
  return <Child onButtonClick={handleClick} />;
};

// Child Component
const Child = (props) => {
  return <button onClick={props.onButtonClick}>Click Me</button>;
};

export default Parent;
```

Managing State

State represents **data that changes** over time.

State is local and private to the component.

State changes cause the **component to re-render.**

For functional components, use the **useState** hook.

React Functions that start with word use are called hooks

Hooks should only be used **inside components**

Parent components can pass state down to children via props.

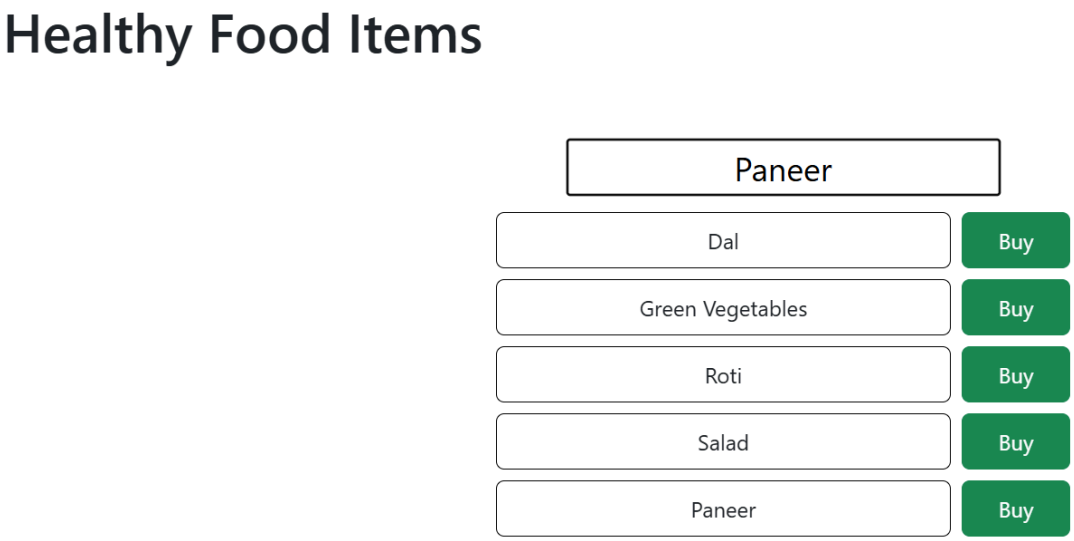
Lifting state up: share state between components by moving it to their closest common ancestor.

Syntax :

```
const [state, setState] = useState(initialValue);
```

```
import Items from './Items'
import { useState } from 'react';

function HealthFood() {

/* State */
  const [foods, setFoods] = useState(['Dal', 'Green Vegetables', 'Roti', 'Salad']);

  function handleByonChange(event){
    if(event.key==="Enter"){
      let newFood =event.target.value;
      let newItems =[...foods,newFood]; /* spread Operator */
      setFoods(newItems) /* Using setFood for updating food */
    }
  }

   return (
```

```
    <>
      <h1>Healthy Food Items</h1>
      <div>

        <input type="text" onKeyDown={handleByonChange}
         placeholder='Enter Food item here' />

      </div>

      <div className="d-flex mx-auto w-50">

        {foods.length === 0 ? <h3>I am Hungry</h3> : null}

        <ul>
          {foods.map((items) => (<Items key={items} food={items} />))}
        </ul>

      </div>
    </>

  )
}

export default HealthFood
```



State vs Props

State	Props
Local and mutable data within a component.	Passed into a component from its parent.
Initialized within the component.	**Read-only (immutable)** within the receiving component.
Can **change** over time	Allow **parent-to-child** component communication.
Causes **re-render** when updated. Managed using **useState** in functional components	Changes in props can also cause a **re-render.**

React-icon Library

You can use a lot of icons without managing them.

Install Package

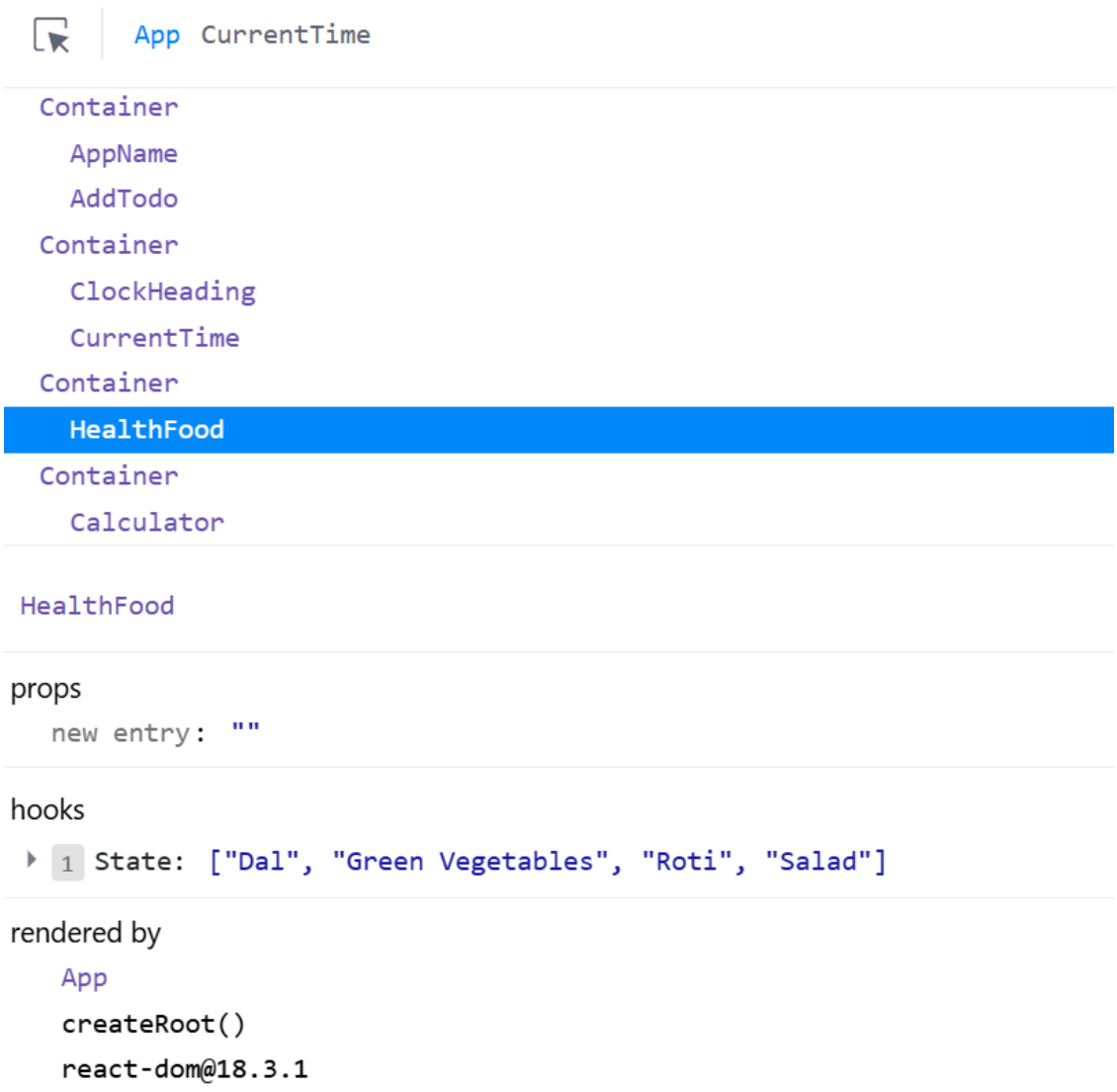
```
npm install react-icons --save
```

usage

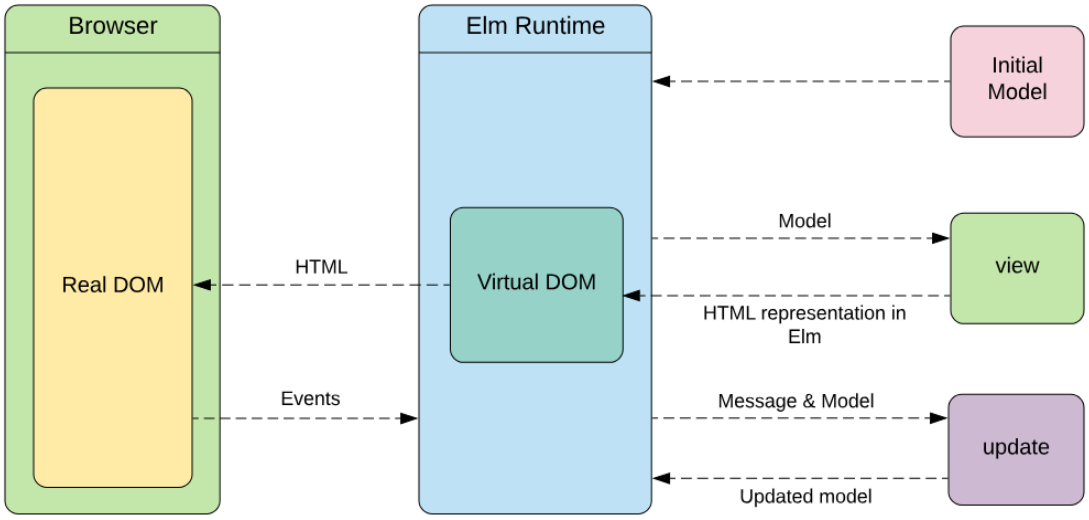
```
import {IconName} from "react-icons/fa";
```

Inspecting with React Dev Tools

The React Developer Tools offer several powerful features for developers. Inspection allows you to examine the hierarchies of React components. With State & Props, you can view and edit the current state and props of components in real-time. The Performance feature helps analyze component re-renders and identify performance bottlenecks. Navigation enables convenient browsing through the entire component tree. Filtering allows you to search for components by name or source, making it easier to locate specific ones quickly. Finally, Real-time Feedback lets you see live changes as you modify the state or props, providing instant visual feedback during development.



How React Works ?

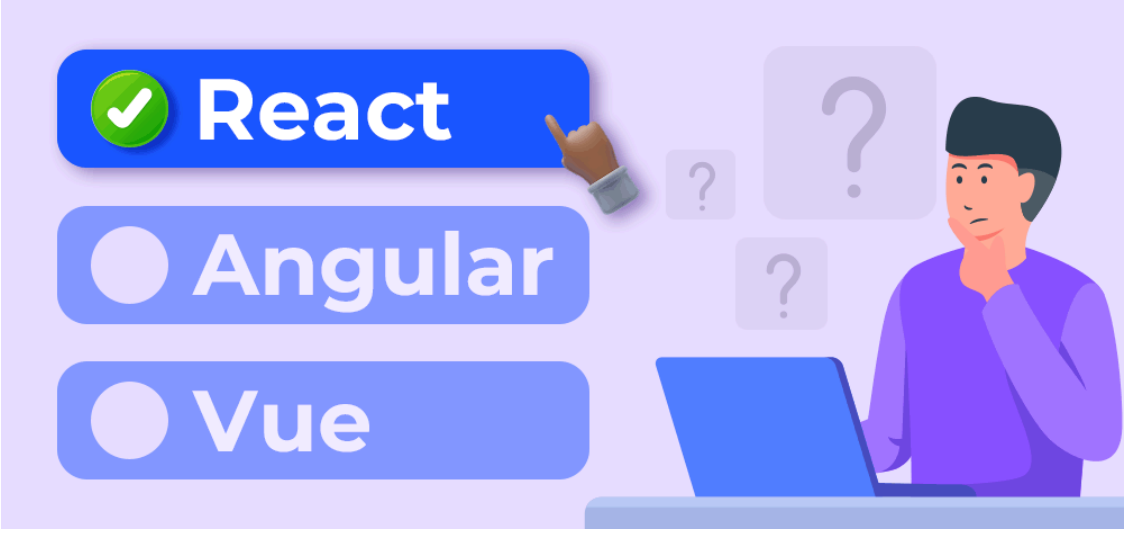


Root Component:
• The App is the main or root component of a React application.
• It's the starting point of your React component tree.

Virtual DOM:
• React creates an in-memory structure called the virtual DOM.
• Different from the actual browser DOM.
• It's a lightweight representation where each node stands for a **component and its attributes.**

Reconciliation Process:
• When component data changes, React updates the virtual DOM's state to mirror these changes.
• React then compares the current and previous versions of the virtual DOM.
• It identifies the specific nodes that need updating.
• Only these nodes are updated in the real browser DOM, making it efficient

React Vs Angular vs Vue



• React is a library, while Angular and Vue.js are frameworks.
• React specializes in UI components, whereas Angular and Vue.js provide comprehensive tools for complete application development.

Library vs. Framework:
• A library provides specific functionality.
• A framework delivers a complete set of tools and guidelines.
• Simply put: React is a single tool, while Angular and Vue.js are complete toolsets.

React's Specialty:
• React excels at building dynamic, interactive user interfaces.
• It doesn't include built-in solutions for routing, HTTP calls, or state management.

React's Flexibility:
• React gives developers freedom in choosing additional tools.
• Teams can select the best solutions for their specific project needs.

Angular and Vue.js:
• Angular, Google's framework, offers robust features but comes with a steeper learning curve.
• Vue.js stands out for its simplicity and seamless integration, making it accessible for beginners.

Using Forms

State Management: Each input's state is stored in the component's state.
Handling Changes: Use onChange to detect input changes.
Submission: Utilize onSubmit for form submissions and prevent default with event.preventDefault().
Validation: Implement custom validation or use third-party libraries.

```
import React, { useState } from "react";

function ControlledForm() {
  const [formData, setFormData] = useState({
    name: "",
    email: "",
  });

  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData({
      ...formData,
      [name]: value,
    });
  };

  const handleSubmit = (e) => {
    e.preventDefault();
```

```
    console.log("Form submitted:", formData);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input
          type="text"
          name="name"
          value={formData.name}
          onChange={handleChange}
        />
      </label>
      <br />
      <label>
        Email:
        <input
          type="email"
          name="email"
          value={formData.email}
          onChange={handleChange}
        />
      </label>
      <br />
      <button type="submit">Submit</button>
    </form>
  );
}

export default ControlledForm;
```

Use Ref

useRef allows access to DOM elements and retains mutable values without re-renders.
Used with the ref attribute for direct DOM interactions.
Can hold previous state or prop values.
Not limited to DOM references; can hold any value.
Refs can be passed as props also

```
import React, { useRef, useState } from "react";

function ControlledForm({newItem}) {
  const formName = useRef();
```

```
  const formMail = useRef();

  const [formData, setFormData] = useState({
    name: "",
    email: "",
  });

  const handleSubmit = (e) => {
    e.preventDefault();
    const fname = formName.current.value;
    const fmail = formMail.current.value;
    formName.current.value="";
    formMail.current.value="";

    setFormData({
       ...formData,
      name: fname,
      email: fmail,
    });

  console.log(formData);

  };


  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input type="text" name="name" ref={formName} placeholder="enter name" />
      </label>
      <br />
      <label>
        Email:
        <input type="email" name="email" ref={formMail} />
      </label>
      <br />
      <button type="submit">Submit</button>
    </form>
  );
}

export default ControlledForm;
```

Update state from Previous State

Spread Operator: Use to maintain immutability when updating arrays or objects.
Functional Updates: to avoid stale values during asynchronous updates.

```
import React, { useRef, useState, useEffect } from "react";

function ControlledForm() {
  const formName = useRef();
  const formMail = useRef();

  const [formData, setFormData] = useState([
    // Initial empty array (or could be pre-populated data)
  ]);

  const handleSubmit = (e) => {
    e.preventDefault();
    const fname = formName.current.value;
    const fmail = formMail.current.value;

    // Clear input fields
    formName.current.value = "";
    formMail.current.value = "";

    // Add new item to formData
    setFormData((prevData) => [
      ...prevData, // Spread previous data
      { name: fname, email: fmail }, // New item
    ]);
  };

  // Log formData when it updates
  useEffect(() => {
    console.log("Updated Form Data:", formData);
  }, [formData]);

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input type="text" name="name" ref={formName} placeholder="Enter name" />
      </label>
      <br />
      <label>
        Email:
```

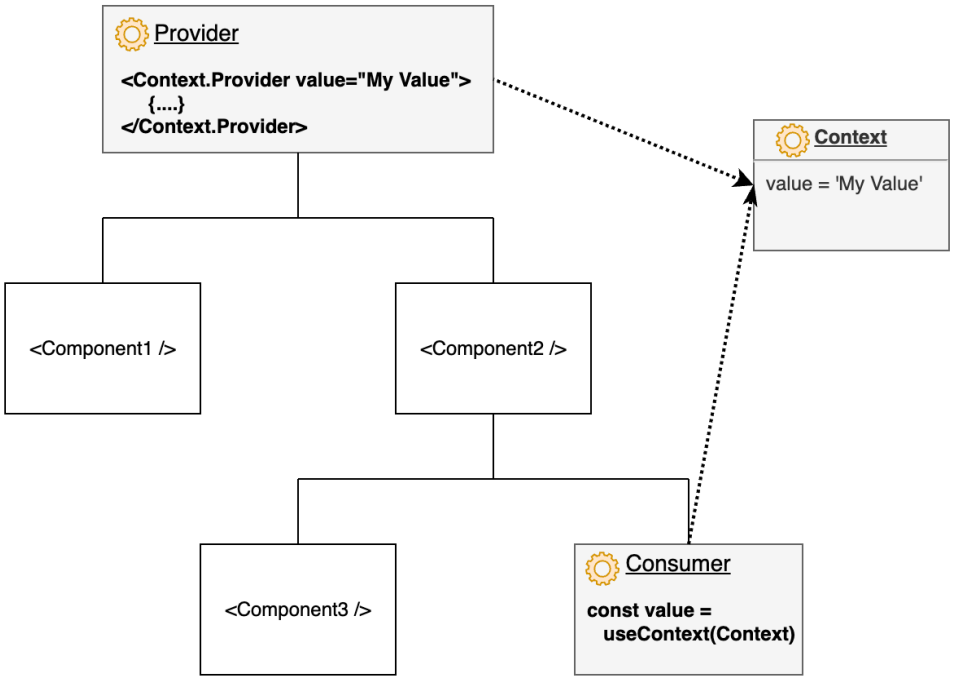
```
      <input type="email" name="email" ref={formMail} />
    </label>
    <br />
    <button type="submit">Submit</button>
  </form>
 );
}

export default ControlledForm;
```

```
Updated Form Data:                    ControlledForm.jsx:29
▼ [{…}] i
  ▶ 0: {name: 'fds', email: 'sandy123@gmail.com'}
    length: 1
  ▶ [[Prototype]]: Array(0)
```

```
Updated Form Data:                    ControlledForm.jsx:29
▼ (2) [{…}, {…}] i
  ▶ 0: {name: 'fds', email: 'sandy123@gmail.com'}
  ▶ 1: {name: 'nf', email: 'fsd@gmail.com'}
    length: 2
  ▶ [[Prototype]]: Array(0)
```

Context API

React Context



Prop Drilling: Context API addresses prop drilling; component composition is an alternative.
Folder Setup: Use a store folder for context files.
Initialization: Use React.createContext with initial state and export it.
Provider: Implement with contextName.Provider in components.
Access Value: Use the useContext hook.
Dynamic Data: Combine context value with state.
Export Functions: Context can also export functions for actions
Logic Separation: This helps keep the UI and business logic separate from each other.

```
// UserContext.js
import React, { createContext, useState } from "react";

// Creating the context
export const UserContext = createContext();

// Creating the provider component
export const UserProvider = ({ children }) => {
```

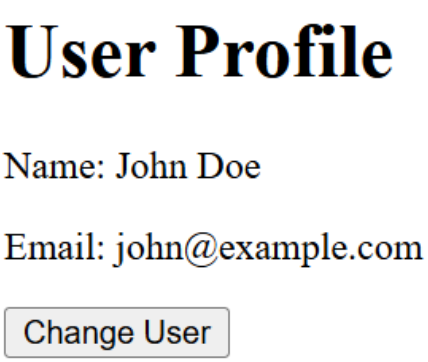
```
  const [user, setUser] = useState({ name: "John Doe", email: "john@example.com" });

  // Function to update user data
  const updateUser = (newUser) => {
    setUser(newUser);
  };

  return (
    <UserContext.Provider value={{ user, updateUser }}>
      {children}
    </UserContext.Provider>
  );
};
```

```
// UserProfile.js
import React, { useContext } from "react";
import { UserContext } from "./UserContext";

const UserProfile = () => {
  const { user, updateUser } = useContext(UserContext);

  // Function to handle user data update
  const handleChangeUser = () => {
    updateUser({ name: "Jane Doe", email: "jane@example.com" });
  };

  return (
    <div>
      <h1>User Profile</h1>
      <p>Name: {user.name}</p>
      <p>Email: {user.email}</p>
      <button onClick={handleChangeUser}>Change User</button>
    </div>
  );
};

export default UserProfile;
```

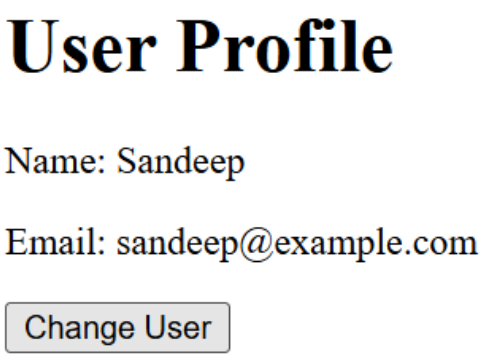
```
//App.jsx
import './App.css'
import { UserProvider } from "./UserContext";
```

```
import UserProfile from "./UserProfile";

function App() {


 return (
  <>

    <UserProvider>
    <div>
      <UserProfile />
    </div>
   </UserProvider>
   </>
 )
}

export default App
```

Before clicking on Change User :

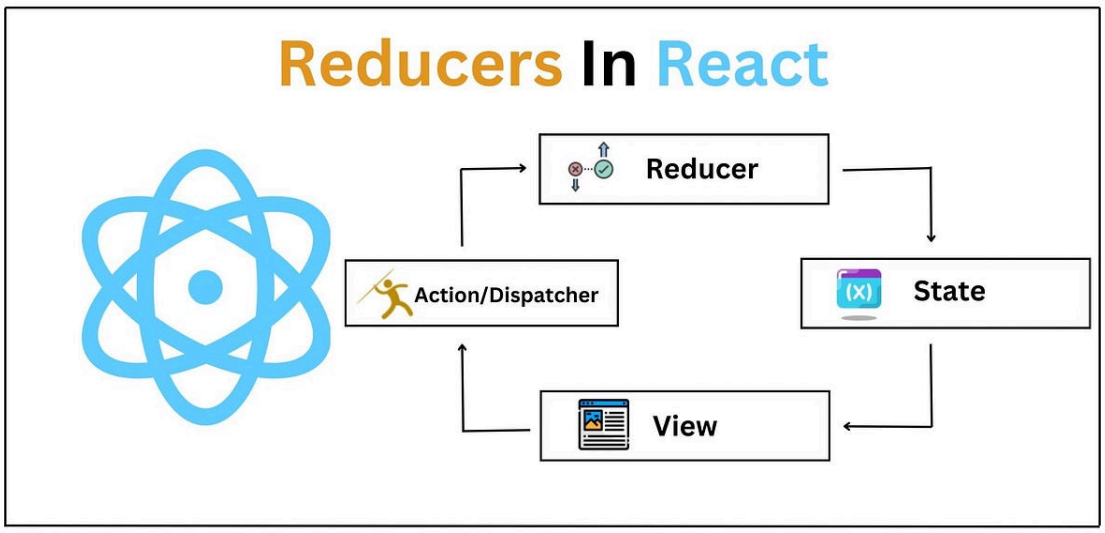


After clicking on Change User :



Use Reducer

1. useReducer is a React hook that provides more control over state management than useState, particularly for complex state logic.
2. Components: It consists of two main parts:
   • Reducer: A pure function that accepts the current state and an action, then returns a new state.
   • Action: An object that describes what happened, typically with a type property.
3. Initialization: It's used with the syntax:
   const [state, dispatch] = useReducer(reducer, initialState).
4. Dispatch: The dispatch function triggers actions, which calls the reducer with the current state and specified action.
5. Use Cases: It's especially valuable for managing state in large components or when new state depends on previous state.
6. Predictable State Management: Its structured approach ensures more predictable and maintainable state management.

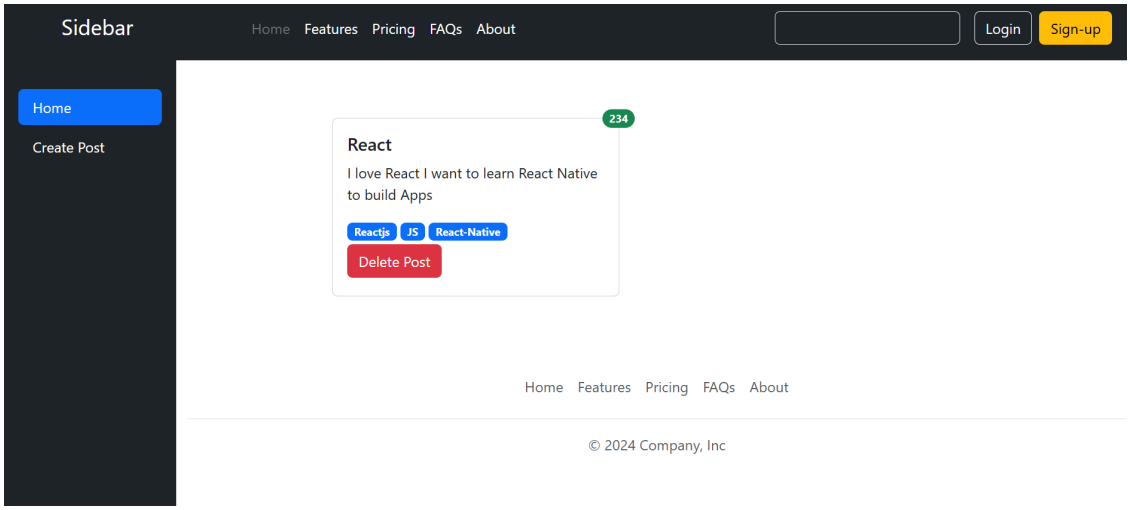



```
import React, { useReducer } from 'react';

// Define the initial state
const initialState = { count: 0 };

// Define the reducer function
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
    </div>
```

```
  );
}

export default Counter;
```

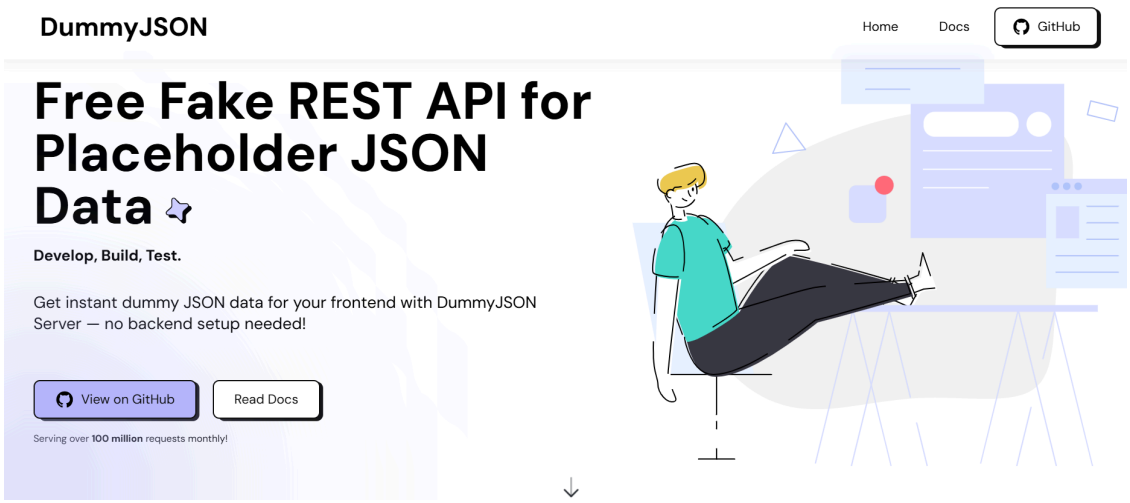
Revision of all the above topics can be done by creating a project including .

Social Media (Project v1)



Dummy JSON

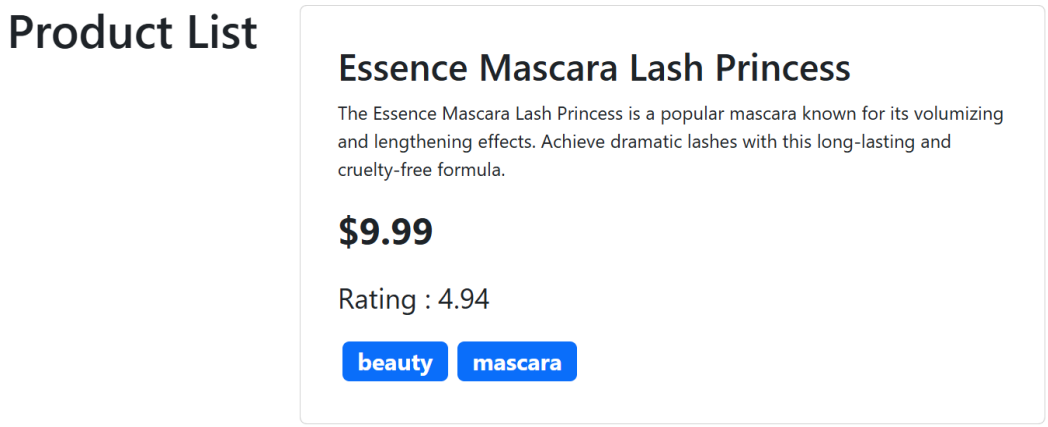
Link to DummyJSON



Data fetching using Fetch

fetch: Modern JavaScript API for network requests.
Promise-Based: Returns a Promise with a Response object.
Usage: Default is GET. For POST use method: 'POST'
Response: Use .then() and response.json() for JSON data.
Errors: Doesn't reject on HTTP errors. Check response.ok.
Headers: Managed using the Headers API.

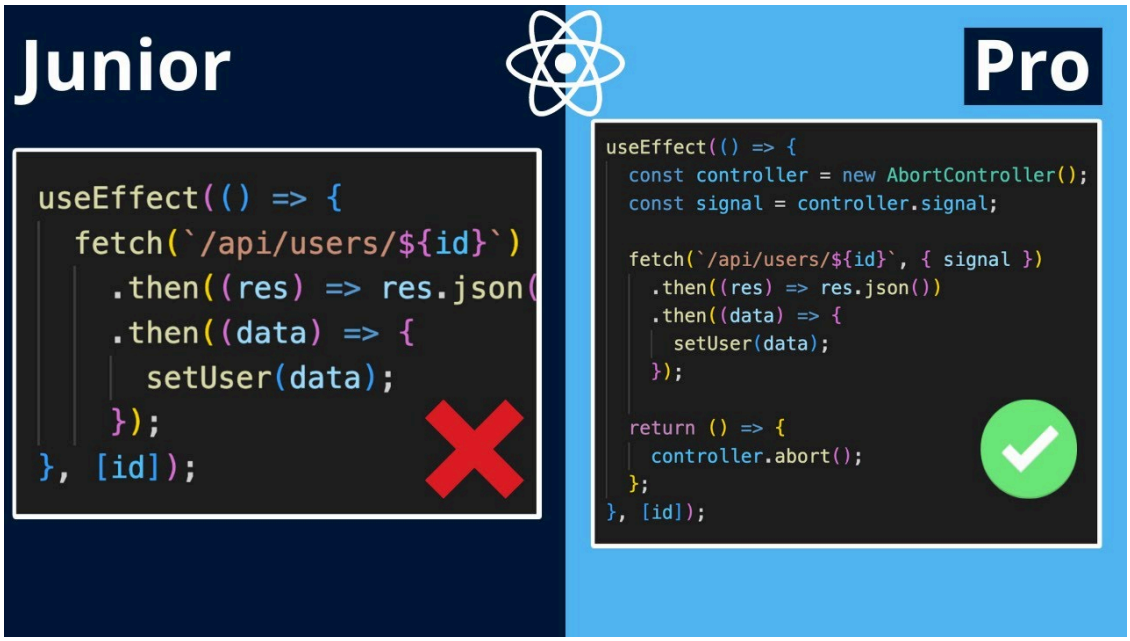
```
fetch('https://dummyjson.com/products')
    .then((res) => res.json())
    .then((data) => setData(data.products)) // Update state with fetched products
    .catch((err) => console.error(err));
```



UseEffect

In function-based components, useEffect handles side effects like data fetching or event listeners.
useEffect runs automatically after every render by default.
By providing a dependency array, useEffect will only run when specified variables change.

An empty array means the effect runs once.
Multiple useEffect hooks can be used in a single component for organizing different side effects separately.




```jsx
import React, { useEffect, useState } from "react";

function CurrentTime() {
 const [time, setTime] = useState(new Date());

 useEffect(() => {
  //Using setInterval function for updating second in Time .
  const intervalId = setInterval(() => {
   setTime(new Date());
  }, 100);

  //if we open use other pages the setInterval function will not work in background
  return () => {
   clearInterval(intervalId);
  };
 });

 return (
  <div>
   <p>This is the clock that shows the time in bharat at all time.</p>
   <p className="fw-bold "> Date : {time.toLocaleDateString()}</p>
   <p className="fw-bold ">
    {" "}
    This is Current Time : {time.toLocaleTimeString()}
   </p>
```

```
    </div>
  );
}

export default CurrentTime;
```

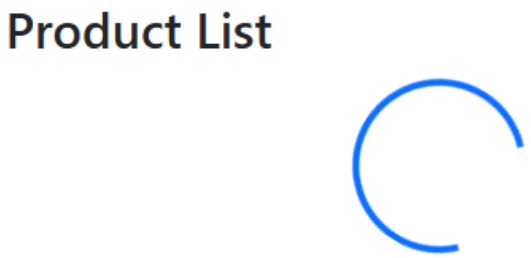
Bharat Clock

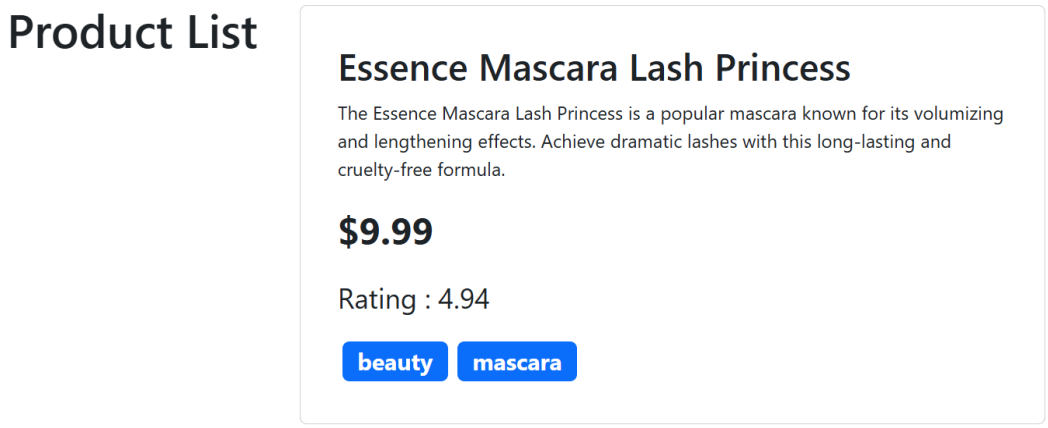
This is the clock that shows the time in bharat at all time.

Date : 12/6/2024

This is Current Time : 7:43:19 PM

Handling Loading State





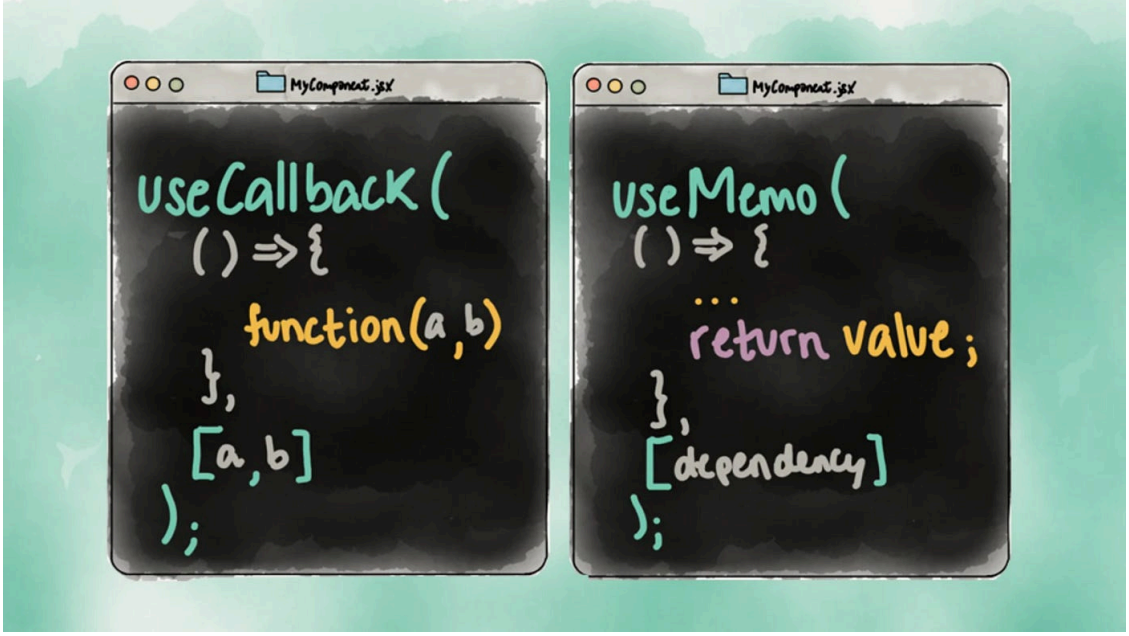
Optimizations Hooks :

The useCallback Hook

Memoization: Preserves function across renders to prevent unnecessary re-renders.
Optimization: Enhances performance in components with frequent updates.
Dependency Array: Recreates the function only when specific dependencies change.
Event Handlers: Used to keep consistent function references for child components.
With useEffect: Prevents infinite loops by maintaining function references

The useMemo Hook

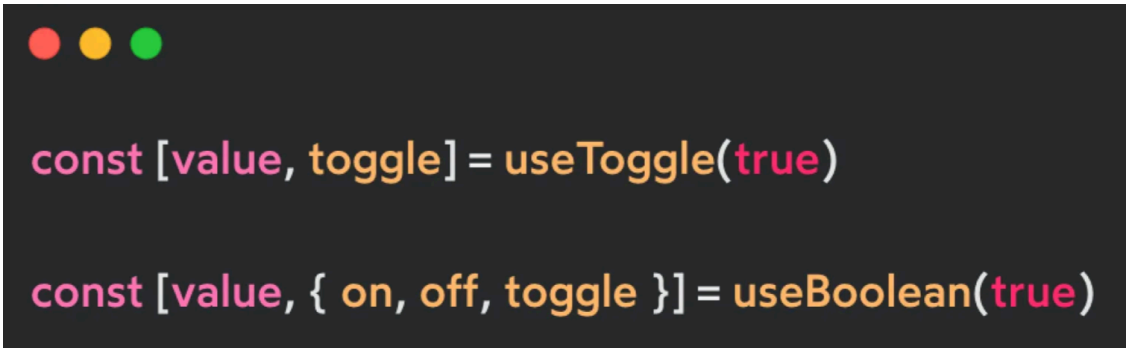
Memoization: useMemo caches the result of expensive calculations to enhance performance. Re-**computation**: Only re-computes the memoized value when specific dependencies change.
Optimization: Helps prevent unnecessary recalculations, improving component rendering efficiency.
Dependency Array: Uses an array of dependencies to determine when to recompute the cached value.
Comparison with useCallback: While useCallback memoizes functions, useMemo memoizes values or results of functions.
Best Use: Ideal for intensive computations or operations that shouldn't run on every render.



Custom Hooks

Reusable Logic: Custom hooks allow you to extract and reuse component logic across multiple components.
Naming Convention: Typically start with "use" (e.g., useWindowSize, useFetch).
Combining Hooks: Custom hooks can combine multiple built-in hooks like useState, useEffect, and others.
Sharing State: Enables sharing of stateful logic without changing component hierarchy.
Isolation: Helps in isolating complex logic, making components cleaner and easier to maintain.

Custom Return Values: Can return any value (arrays, objects, or any other data type) based on requirements.



Submitting data with fetch

```
import React, { useState } from "react";

function SubmittingData() {
  const [formData, setFormData] = useState({
    name: "",
    email: "",
    message: "",
  });

  const [responseMessage, setResponseMessage] = useState("");

  // Handle input changes
  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData((prevData) => ({
      ...prevData,
      [name]: value,
    }));
  };

  // Handle form submission
  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      const response = await fetch("https://dummyjson.com/posts/add", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify(formData),
      });

      if (response.ok) {
        const result = await response.json();
        setResponseMessage("Form submitted successfully: " + result.message);
      } else {
        setResponseMessage("Error submitting form: " + response.statusText);
      }
    } catch (error) {
      setResponseMessage("Network error: " + error.message);
    }
```

```
  };

  return (
    <div>
      <h1>Submit Data with Fetch in React</h1>
      <form onSubmit={handleSubmit}>
        <div>
          <label>
            Name:
            <input
              type="text"
              name="name"
              value={formData.name}
              onChange={handleChange}
              required
            />
          </label>
        </div>
        <div>
          <label>
            Email:
            <input
              type="email"
              name="email"
              value={formData.email}
              onChange={handleChange}
              required
            />
          </label>
        </div>
        <div>
          <label>
            Message:
            <textarea
              name="message"
              value={formData.message}
              onChange={handleChange}
              required
            ></textarea>
          </label>
        </div>
        <button type="submit">Submit</button>
      </form>
      {responseMessage && <p>{responseMessage}</p>}
    </div>
```

```
    );
}

export default SubmittingData;
```

React Router

Installation: Use npm install react-router-dom.
We are going to use the latest version which is 6+
RouterProvider: Wraps the app for routing capabilities.
createBrowserRouter: helps creating the mapping for router provider.
Declarative Routing: Easily define application routes.
Routes are React components.

Layout Routes help us to use shared elements
Outlet component is used to render the children at the correct places

Router Layout with React Router

```
import React from "react";
import {
  BrowserRouter as Router,
  Routes,
  Route,
  Outlet,
  Link,
} from "react-router-dom";

// App Component with Router Setup
const App = () => {
  return (
    <Router>
      <Routes>
        {/* Layout applied to these routes */}
        <Route path="/" element={<Layout />}>
          <Route index element={<Home />} />
          <Route path="about" element={<About />} />
          <Route path="contact" element={<Contact />} />
        </Route>
      </Routes>
    </Router>
  );
```

```
};

export default App;
```

Using Object-Based Route Definitions

```
import React from "react";
import { BrowserRouter as Router, useRoutes } from "react-router-dom";

const AppRoutes = () => {
  const routes = [
    { path: "/", element: <Home /> },
    { path: "/about", element: <About /> },
  ];

  return useRoutes(routes);
};

const App = () => {
  return (
    <Router>
      <AppRoutes />
    </Router>
  );
};

export default App;
```

Route Links

Link Component with to property can be used to avoid reloading
useNavigate hook can be used to do navigation programmatically

`Link` Component

```
import React from "react";
import { BrowserRouter as Router, Routes, Route, Link } from "react-router-dom";

const Home = () => <h1>Home Page</h1>;
```

```
const About = () => <h1>About Page</h1>;

const App = () => {
 return (
  <Router>
   <nav>
    <ul>
     <li>
      <Link to="/">Home</Link>
     </li>
     <li>
      <Link to="/about">About</Link>
     </li>
    </ul>
   </nav>

   <Routes>
    <Route path="/" element={<Home />} />
    <Route path="/about" element={<About />} />
   </Routes>
  </Router>
 );
};

export default App;
```

`useNavigate` Hook

```
import React from "react";
import { BrowserRouter as Router, Routes, Route, useNavigate } from "react-router-dom";

const Home = () => {
 const navigate = useNavigate();

 const handleGoToAbout = () => {
  navigate("/about");
 };

 return (
  <div>
   <h1>Home Page</h1>
   <button onClick={handleGoToAbout}>Go to About</button>
```

```
    </div>
  );
};

const About = () => <h1>About Page</h1>;

const App = () => {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </Router>
  );
};

export default App;
```

Data fetching using loader

Loader method can be used to load
data before a particular route is
executed.
The loader method must return the data that is loaded or promise.
Data is available in component and all the child components.
useLoaderData hook can be used to get the fetched data.
Loading state can also be used.

s

```
import React from "react";
import {
  createBrowserRouter,
  RouterProvider,
  useLoaderData,
  Outlet,
} from "react-router-dom";

// Loaders for different routes
const fetchPosts = async () => {
  const response = await fetch("https://jsonplaceholder.typicode.com/posts");
```

```
  return response.json();
};

const fetchPostDetails = async ({ params }) => {
  const response = await fetch(
    `https://jsonplaceholder.typicode.com/posts/${params.postId}`
  );
  return response.json();
};

// Components
const Layout = () => (
  <div>
    <h1>Blog</h1>
    <Outlet /> {/* Renders child routes here */}
  </div>
);

const Posts = () => {
  const posts = useLoaderData();

  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>
          <a href={`/post/${post.id}`}>{post.title}</a>
        </li>
      ))}
    </ul>
  );
};

const PostDetails = () => {
  const post = useLoaderData();

  return (
    <div>
      <h2>{post.title}</h2>
      <p>{post.body}</p>
    </div>
  );
};

// Create the router
const router = createBrowserRouter([
```

```
  {
    path: "/",
    element: <Layout />,
    children: [
      {
        path: "/",
        element: <Posts />,
        loader: fetchPosts,
      },
      {
        path: "/post/:postId",
        element: <PostDetails />,
        loader: fetchPostDetails,
      },
    ],
  },
]);

const App = () => {
  return <RouterProvider router={router} />;
};

export default App;
```

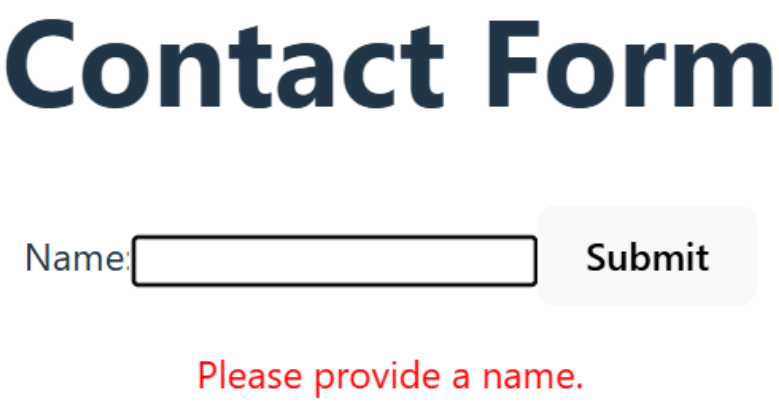
Submitting data using action

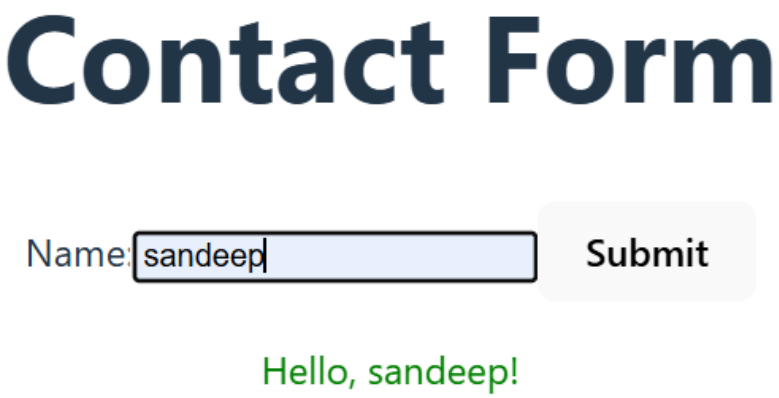
Action method can be used to perform an action on submission of Forms.
Custom Form component need to be used along with name attribute for all inputs.
Action function will get an data object. To generate correct request object method="post" attribute should be used.
Data.request.formData() method can be used to get form data Object.
Object.fromEntries(formData) can be used to get actual input data.
redirect() response can be returned for navigation after submission

```
import React from "react";
import {
    createBrowserRouter,
    RouterProvider,
    useActionData,
    Form,
```

```
} from "react-router-dom";

// Action function to handle form submission
const submitData = async ({ request }) => {
  const formData = await request.formData(); // Parse submitted form data
  const name = formData.get("name");


  // Example logic: You can send this data to a server or process it locally
  if (name) {
    return { success: true, message: `Hello, ${name}!` };
  }
  return { success: false, message: "Please provide a name." };
};

// Component to display the form and submission results
const ContactForm = () => {
  const actionData = useActionData(); // Access the result of the action

  return (
    <div style={{ margin: "24px", textAlign: "center" }}>
      <h1>Contact Form</h1>
      <Form method="post">
        <label>
          Name:
          <input type="text" name="name" />
        </label>
        <button type="submit">Submit</button>
      </Form>
      {actionData && (
        <p style={{ color: actionData.success ? "green" : "red" }}>
          {actionData.message}
        </p>
      )}
    </div>
  );
};

// Router setup
const router = createBrowserRouter([
  {
    path: "/",
    element: <ContactForm />,
    action: submitData, // Attach the action function
  },
```

```
]);

const App = () => {
  return <RouterProvider router={router} />;
};

export default App;
```

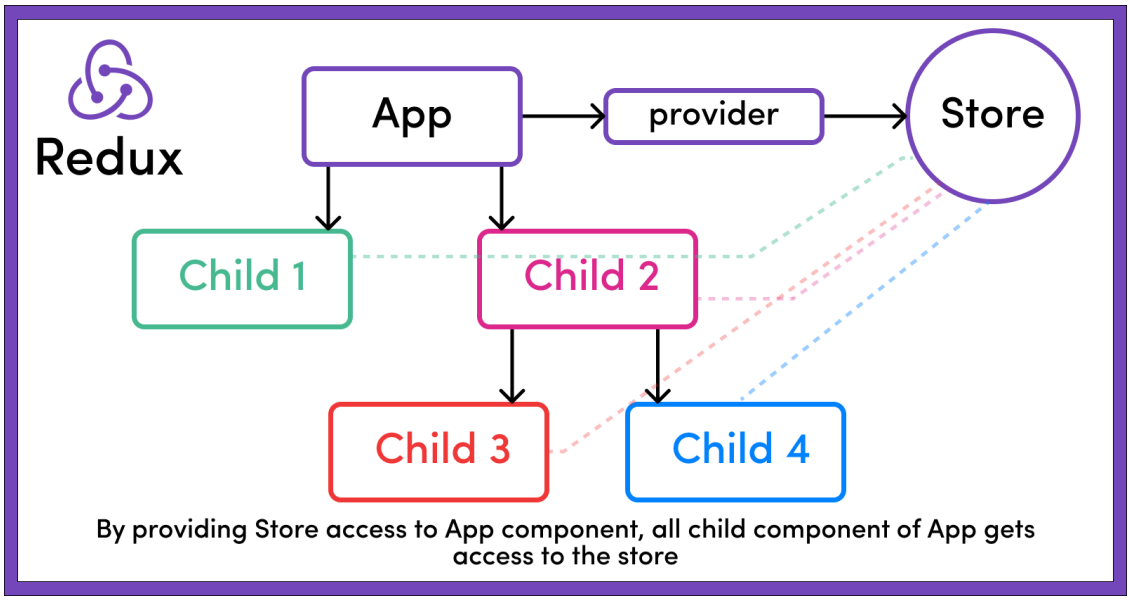




What is Redux

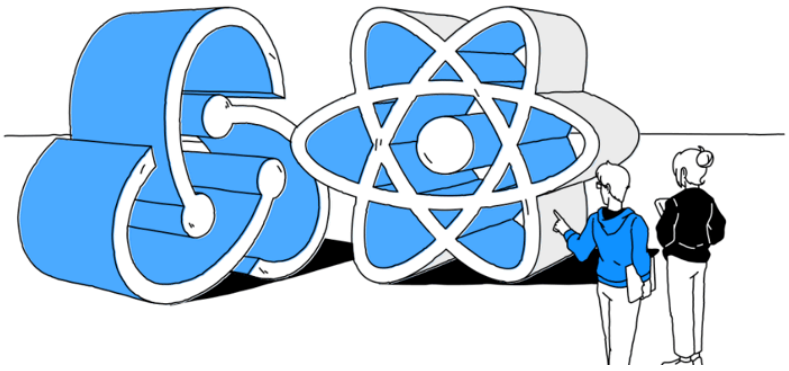
State management for cross component or app-wide state.
Redux is a predictable state management library for JavaScript apps.

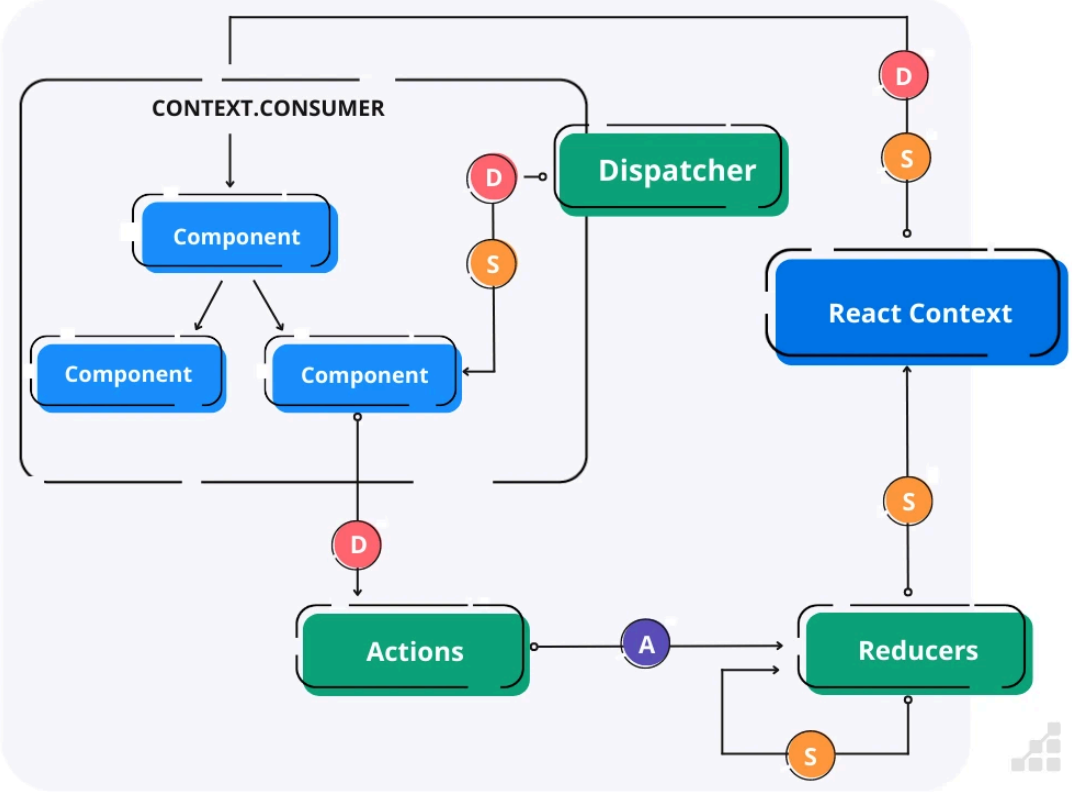
Local State vs Cross-component state vs App-Wide state
useState or useReducer vs useState with prop drilling vs useState or useContext or Redux



React-Context VS Redux

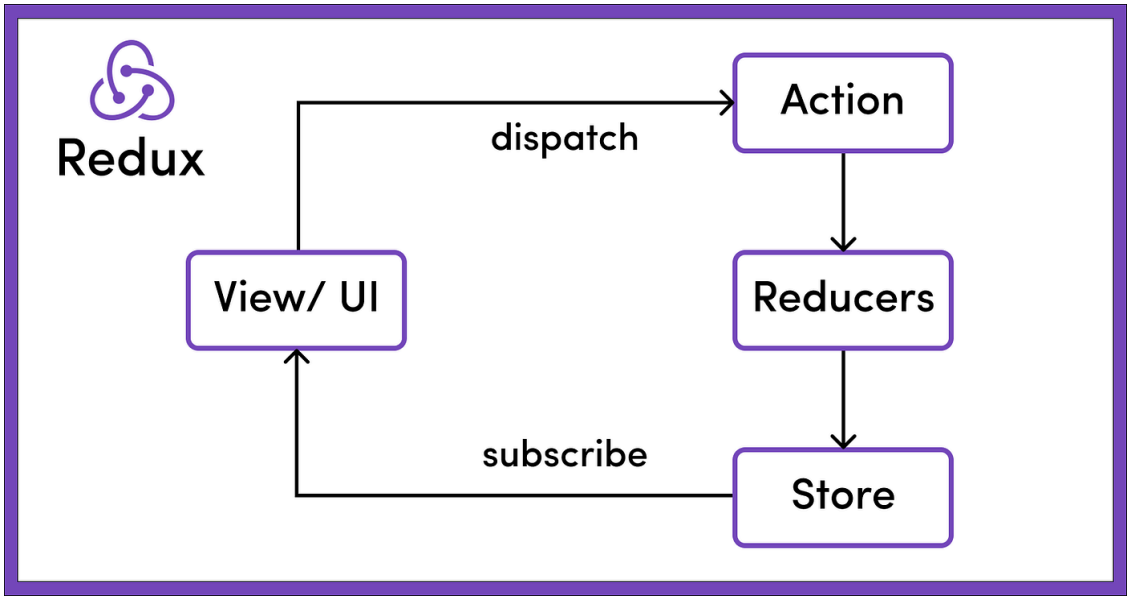
You can use both.
Setup and Coding is tough especially if you have multiple context providers.
Performance is slow. Context should only be used for things that rarely change. On the other hand Redux has great performance.
If these things don't matter to you then you can choose not to use redux and stay with React-Context.





How Redux Works

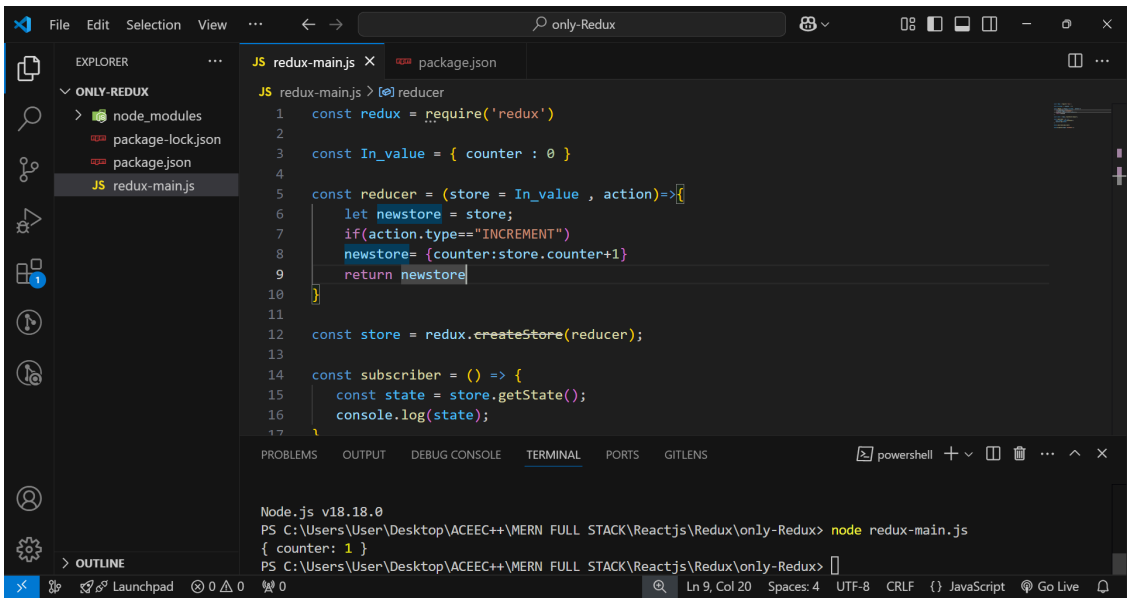
1. Single Source: Uses a single central store to maintain the entire application's state.
2. Actions: Components never directly change the store. Changes to state are made through dispatched actions, which describe events.
3. Reducers: Actions are processed by reducers, pure functions that return the new state.
4. Immutable: State is immutable; every change results in a new state object.
5. This is different from useReducer hook but little similar to it.



Working with Redux

1. npm init–y
2. npm install redux
3. import in node Const redux = require('redux');
4. We need to setup all 4 basic things:
5. Reducer
6. Store
7. Subscriber
8. Actions
9. Node redux-demo.js command to run node serve

```
const redux = require('redux')

const In_value = { counter : 0 }

const reducer = (store = In_value , action)=>{
   let newstore = store;
   if(action.type=="INCREMENT")
   newstore= {counter:store.counter+1}
   return newstore
}

const store = redux.createStore(reducer);

const subscriber = () => {
  const state = store.getState();
  console.log(state);
}

store.subscribe(subscriber)

store.dispatch({type:'INCREMENT'});
```



React with Redux

1. Npm install redux
2. Npm install react-redux
3. Create store folder with Index.js file
4. Creating the store using
   Import {createStore} from redux.
5. Providing the store with react
6. Provider from react-redux
7. **<Provider store={store}><App /></Provider>**
8. Using the store
9. useSelector hook gets a slice of the store.
   Const counter = useSelector(state => state.counter);
10. Subscription is already setup and only will re-execute when only your slice is changed. Subscription is automatically cleared also.
11. Dispatch Actions using the useDispatch hook

Counter Example :

```
// store/index.js
import { createStore } from "redux";

const INITIAL_VALUE = {
  counter: 0,
  privacy: false,
};

const counterReducer = (store = INITIAL_VALUE, action) => {
  if (action.type == "INCREMENT") {
    return { counter: store.counter + 1, privacy: store.privacy };
  } else if (action.type == "DECREMENT") {
    return { counter: store.counter - 1, privacy: store.privacy };
  } else if (action.type == "ADD") {
    return {
      counter: store.counter + Number(action.payload.num),
```

```
      privacy: store.privacy,
    };
  } else if (action.type == "SUB") {
    return {
      counter: store.counter - Number(action.payload.num),
      privacy: store.privacy,
    };
  } else if (action.type == "PRIVACYTOGGLE") {
    return { counter: store.counter, privacy: !store.privacy };
  }
  return store;
};

const counterStore = createStore(counterReducer);

export default counterStore;
```

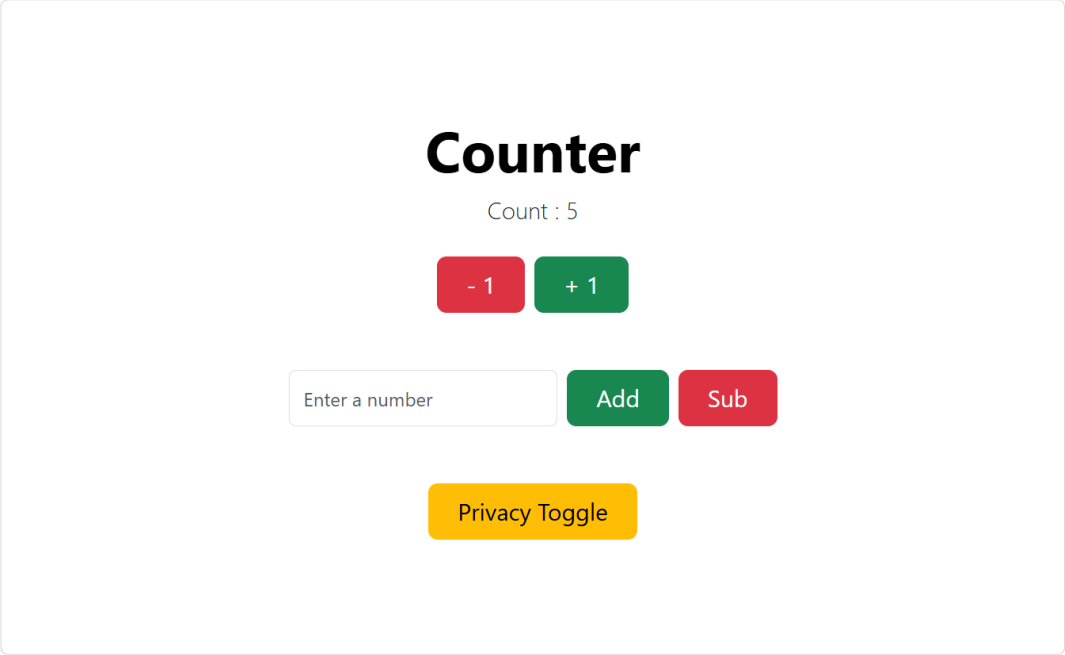
```
// components/Counter.jsx
import React from 'react'
import { useSelector } from 'react-redux'

function Counter() {
  const counter = useSelector(store => store.counter)
  return (
    <div>
      <p>Count : {counter}</p>
    </div>
  )
}

export default Counter
```

```
// components/Controlls.jsx
import React, { useRef } from "react";
import { useDispatch } from "react-redux";

function Controlls() {
  const dispatch = useDispatch();
  const inputElement = useRef();

  const handleIncrement = () => {
```

```
  dispatch({ type: "INCREMENT" });
};

const handleDecrement = () => {
  dispatch({ type: "DECREMENT" });
};

const handleAdd = () => {
  dispatch({
    type: "ADD",
    payload: {
      num: inputElement.current.value,
    },
  });

  inputElement.current.value = "";
};

const handleSub = () => {
  dispatch({
    type: "SUB",
    payload: {
      num: inputElement.current.value,
    },
  });

  inputElement.current.value = "";
};

const handlePrivacyToggle = () => {
  dispatch({
    type: "PRIVACYTOGGLE",
    payload: {
      num: inputElement.current.value,
    },
  });

  inputElement.current.value = "";
};

return (
  <>
    <div class="d-grid gap-2 d-sm-flex justify-content-sm-center">
      <button
        type="button
```

```
        class="btn btn-danger btn-lg px-4 gap-3"
        onClick={handleDecrement}
      >
        {" "}
        - 1{" "}
      </button>
      <button
        type="button"
        class="btn btn-success btn-lg px-4"
        onClick={handleIncrement}
      >
        {" "}
        + 1{" "}
      </button>
    </div>

    <div class="d-grid gap-2 d-sm-flex justify-content-sm-center mt-5">
      <input
        type="text"
        ref={inputElement}
        class="form-control"
        id="exampleInputEmail1"
        aria-describedby="emailHelp"
        placeholder="Enter a number"
      />
      <button
        type="button"
        class="btn btn-success btn-lg px-4"
        onClick={handleAdd}
      >
        {" "}
        Add
      </button>
      <button
        type="button"
        class="btn btn-danger btn-lg px-4"
        onClick={handleSub}
      >
        {" "}
        Sub
      </button>
    </div>
    <div class="d-grid gap-2 d-sm-flex justify-content-sm-center mt-5">
      <button
        type="button"
```

```
        class="btn btn-warning btn-lg px-4"
        onClick={handlePrivacyToggle}
      >
        {" "}
        Privacy Toggle
      </button>
    </div>
  </>
 );
}

export default Controlls;
```



Why Redux Toolkit

1. Action types are difficult to maintain
2. Store becoming too big
3. Mistakenly editing store
4. Reducer becoming too big

Working with Redux Toolkit

1. Npminstall @reduxjs/toolkit
2. Remove redux from package.json
3. Import {createSlice} from “@reduxjs/toolkit”
4. Slices of the store can be created using the following syntax:

```
Const slice = createSlice({
name: '',
initialState: {},
reducers: {
smallReducerMethods: (state, action) => {},
}
})
```

5. ConfigureStore combines multiple reducers and can be used as:

```
configureStore({
reducer: {name: slice.reducer}
})
```

1. Export actions = slice.actions;
2. Actions can be dispatched like: actions.reducerMethod(payload);

Counter using Redux Tool-Kit

```
// store/index.js

import { configureStore } from "@reduxjs/toolkit";
import counterSlice from "./counter";
import privacySlice from "./privacy";
```

```
const counterStore = configureStore({
   reducer: {
      counter: counterSlice.reducer,
      privacy: privacySlice.reducer
   }
})

export default counterStore
```

```
// store/counter.js

import { createSlice } from "@reduxjs/toolkit";

const counterSlice = createSlice({
   name: 'counter',
   initialState: { counterVal: 0 },
   reducers: {
      increment: (state, action) => { state.counterVal++; },
      decrement: (state, action) => { state.counterVal--; },
      add: (state, action) => { state.counterVal += Number(action.payload.num) },
      sub: (state, action) => { state.counterVal -= Number(action.payload.num) },
   }
})

export const counterActions = counterSlice.actions
export default counterSlice
```

```
// store/privacy.js

import { createSlice } from "@reduxjs/toolkit"

const privacySlice = createSlice({
   name: 'privacy',
   initialState: false,
   reducers: {
      toggle: (state) => {return state = !state }
   }
})
```

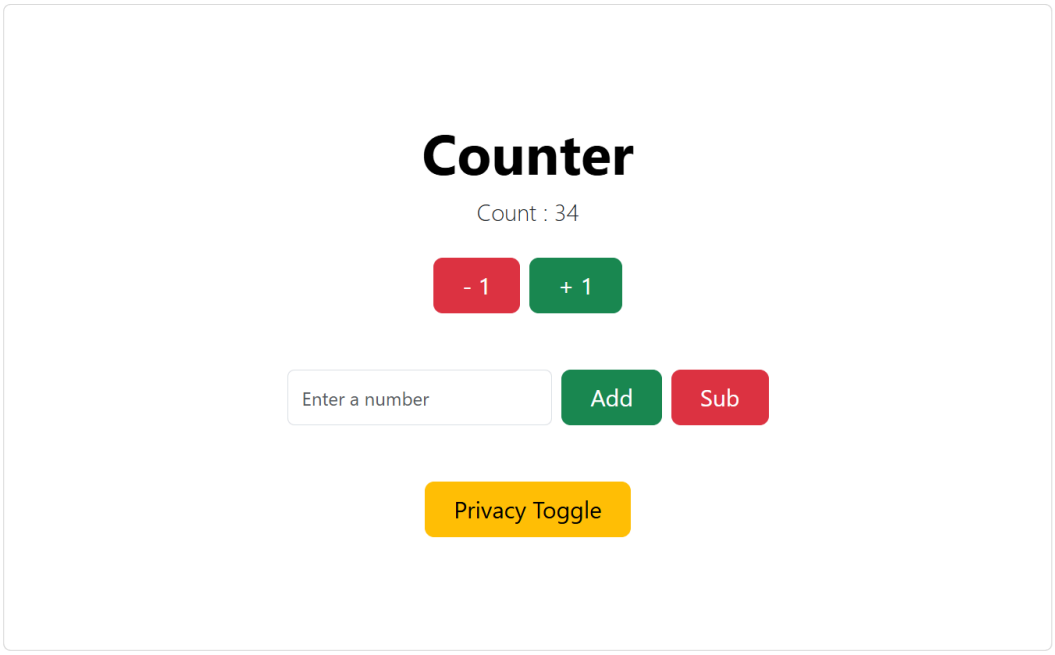
```
export const privacyActions = privacySlice.actions
export default privacySlice
```

```
// components/Cotrolls.jsx

import React, { useRef } from "react";
import { counterActions, privacyActions } from "../store";

function Controlls() {
 // const dispatch = useDispatch()
 const inputElement = useRef();

 const handleIncrement = () => {
  counterActions.increment();
 };

 const handleDecrement = () => {
  console.log("deckdajfdk");
  counterActions.decrement();
 };

 const handleAdd = () => {
  dispatch(
   counterActions.add({
    num: inputElement.current.value,
   })
  );

  inputElement.current.value = "";
 };

 const handleSub = () => {
  dispatch(
   counterActions.sub({
    num: inputElement.current.value,
   })
  );

  inputElement.current.value = "";
 };

 const handlePrivacyToggle = () => {
  dispatch(privacyActions.toggle());
```

```
  inputElement.current.value = "";
};

return (
  <>
    <div class="d-grid gap-2 d-sm-flex justify-content-sm-center">
      <button
        type="button"
        class="btn btn-danger btn-lg px-4 gap-3"
        onClick={handleDecrement}
      >
        {" "}
        - 1{" "}
      </button>
      <button
        type="button"
        class="btn btn-success btn-lg px-4"
        onClick={handleIncrement}
      >
        {" "}
        + 1{" "}
      </button>
    </div>

    <div class="d-grid gap-2 d-sm-flex justify-content-sm-center mt-5">
      <input
        type="text"
        ref={inputElement}
        class="form-control"
        id="exampleInputEmail1"
        aria-describedby="emailHelp"
        placeholder="Enter a number"
      />
      <button
        type="button"
        class="btn btn-success btn-lg px-4"
        onClick={handleAdd}
      >
        {" "}
        Add
      </button>
      <button
        type="button"
        class="btn btn-danger btn-lg px-4"
```

```
        onClick={handleSub}
      >
        {" "}
        Sub
      </button>
    </div>
    <div class="d-grid gap-2 d-sm-flex justify-content-sm-center mt-5">
      <button
        type="button"
        class="btn btn-warning btn-lg px-4"
        onClick={handlePrivacyToggle}
      >
        {" "}
        Privacy Toggle
      </button>
    </div>
  </>
 );
}

export default Controlls;
```

```
// components/Counter.jsx

import React from 'react'
import { useSelector } from 'react-redux'

function Counter() {
 const {counterVal} = useSelector(store => store.counter)
 return (
  <div>
   <p>Count : {counterVal}</p>
  </div>
 )
}

export default Counter
```



🚀🔥 React & Redux Complete Course (2024) with Projects | Notes | Free Certification

For MERN stack admission queries, message us or WhatsApp on +91-8000121313

- GitHub Code Repo: https://github.com/Complete-Coding/React_Complete_YouTube

https://youtu.be/eILUmCJhl64?si=U-heRnMJsrkkZJJy